Convolutional Neural Networks for Driver Classification

Sam Miller

August 30, 2017

1 Introduction

Distracted driving is now a big problem. In the USA, it's estimated that distracted drivers caused 3,477 deaths and 391,000 injuries in 2015 alone (of Transportation [2017]). Therefore there is a lot of potential benefit from eliminating distraction - it would cut road accidents by roughly 20% (for Disease Control and Prevention [2017]).

Fortunately a solution may be close due to the rise of in-car cameras and advances in Computer Vision. Cameras could monitor drivers and, if they are distracted, warn them to focus on the road. This process would need to be automatic, it would not be feasible to have people checking all the images in real time, so we would need an algorithm that a car could use to determine whether a driver was distracted or not from a photo.

In this project I implement a Machine Learning algorithm that learns to determine whether images of drivers are distracted. There are ten different classes of image, comprised of 1 class that is driving safely and 9 classes that are distracted. The algorithm outputs a probability of each class for a given input image.

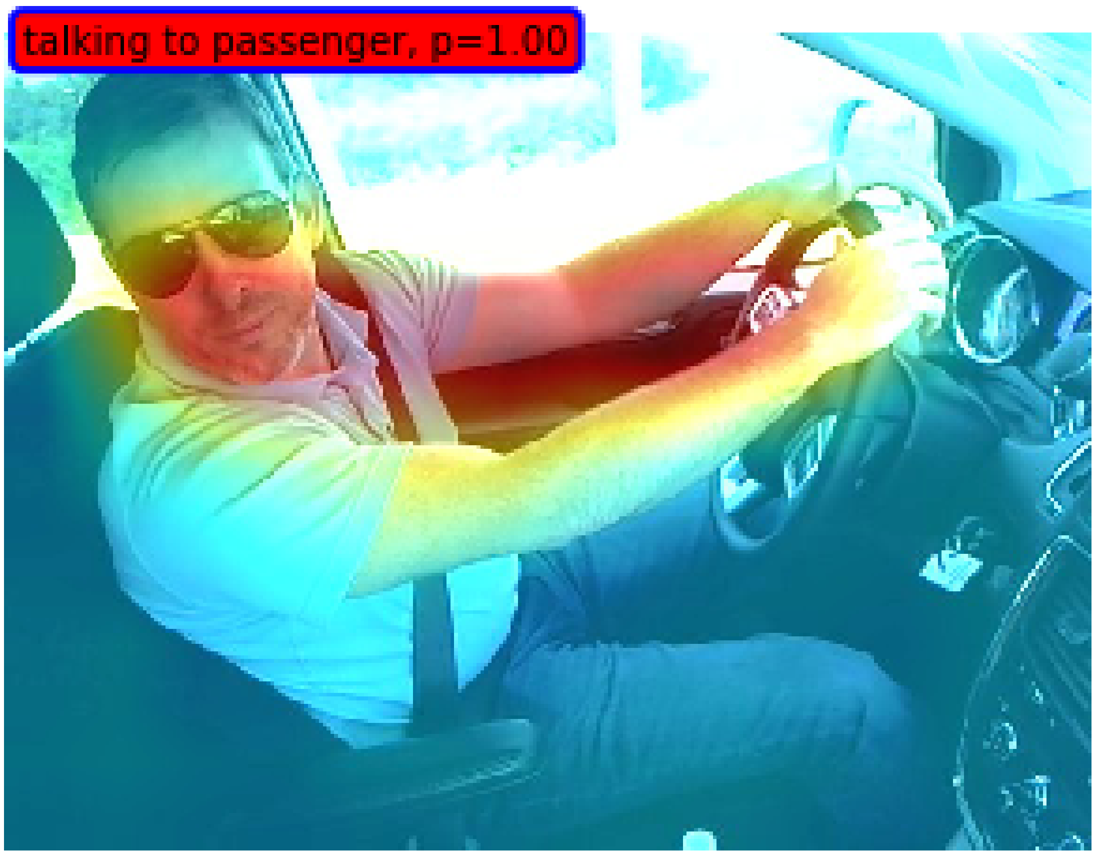
I use a dataset of over 20,000 images to train the algorithm, and then test its performance using nearly 80,000 images. The source of both datasets is Kaggle. I find that a simple Convolutional Neural Network (CNN), based on the VGG16 structure, from Simonyan and Zisserman [2014], has an accuracy of nearly 90%, which greatly improves on a benchmark accuracy of around 10% (bearing in mind there are 10 classes).

We go beyond just classifying images by implementing a Class Activation Map (CAM) that allows us to our classifier's decisions. 1 provides an example of a CAM - the "hotter" areas are where the classifier is placing most weight in making its decision. However we are unable to fine-tune our classifier due to computational restrictions, therefore suggest future improvements for both the algorithm and expanding the dataset.

2 Relevant Literature

Neural Networks attempt to solve complex problems, such as image recognition, by mimicking the human brain. The idea of replicating the smallest unit - the Neuron - dates back to Frank Rosenblatt's "Perceptron" from 1957 (Rosenblatt [1958]. The Perceptron would be initialized with some random weights. It then received some binary inputs and would output 1 if the weighted sum of the inputs exceeded a certain size, but 0 otherwise. If the output was correct then the weights on the "1" inputs would increase and the weights on the "0" inputs would decrease. A single layer of Perceptrons could learn to accurately classify some problems but only if they were linearly separable (Minsky and Papert [1969], so this field of research temporarily died.

Figure 1: Example of a CAM demonstrating the classifier's focus



Neural Networks were revived in 1974 when it was found that they could solve more complex problems by stacking together multiple layers (Werbos [1974]). There would be a single Input Layer, an Output Layer, and at least 1 Hidden Layer in between. The Hidden Layer(s) would pick up basic features of the raw data, then later layers could make decisions based on these features that would be vastly simpler than deciding based on raw data alone. However this discovery did not get much attention until 1986, when Rumelhart et al. [1986] showed that the weights in many layers could be trained via a "backpropagation" algorithm.

LeCun et al. [1989] quickly applied this discovery in 1989 to image recognition. They showed that neural networks could learn to accurately classify handwritten digits. Their main innovation was using Convolutional Layers to massively reduce the number of weights needed to be trained - we provide more details on this in a later section on our model. Despite these theoretical advances Neural Networks again went out of fashion temporarily, although for computational rather than theoretical limits this time.

The next revival was in 2009 when it was shown that training neural networks on a GPU, rather than CPU, could be up to 70 times faster (Raina et al. [2009]). This advance came to the fore in 2012, when the "AlexNet" CNN vastly outperformed all competitors on the ImageNet computer vision challenge Krizhevsky et al. [2012]. This was arguably the beginning of the modern era of using deep neural networks for computer vision. Since then computer vision has focused on improving the performance of CNNs, mostly by increasing their depth and inventing creative new structure. For this project we use the VGG16 structure from Simonyan and Zisserman [2014], which innovates on the AlexNet by using a "deep and simple" structure of stacking many Convolutional layers. This structure has been found to generalise well to other image recognition tasks than the ImageNet challenge.

More recently there has been work on interpreting the decisions of CNNs. Firstly Zhou et al. [2016] showed that one could create a Class Activation Mapping (CAM) of the predictions made by CNNs without any fully connected layers, where "hotter" areas of the image get the most weight. Subsequently Selvaraju et al. [2016] showed that these CAMs could be generalised using a method called Gradient-based CAMs (GradCAMs) to CNNs with fully connected layers. I make use of this generalisation in my paper.

3 Data and Performance Metrics

3.1 Dataset

The dataset is from Kaggle, which is a website that hosts machine learning competitions. There are 22,242 labelled color photos taken from inside various vehicles. All photos are labelled with which distracting activity, if any, the driver is partaking in. The entire feature space will come from the photos alone I will not use any other information to help classify the images, such as dates or times, in order to make this a pure computer vision project. I will use a Python package called Open Computer Vision (OpenCV) to convert the photos to numerical features that a machine learning algorithm can read. OpenCV will break down each photos by its component pixels and assign 3 values to each pixel according to its position on the Red-GreenBlue (RGB) spectrum. The total number of features for each photo will therefore be 3*Height*Width (in pixels).

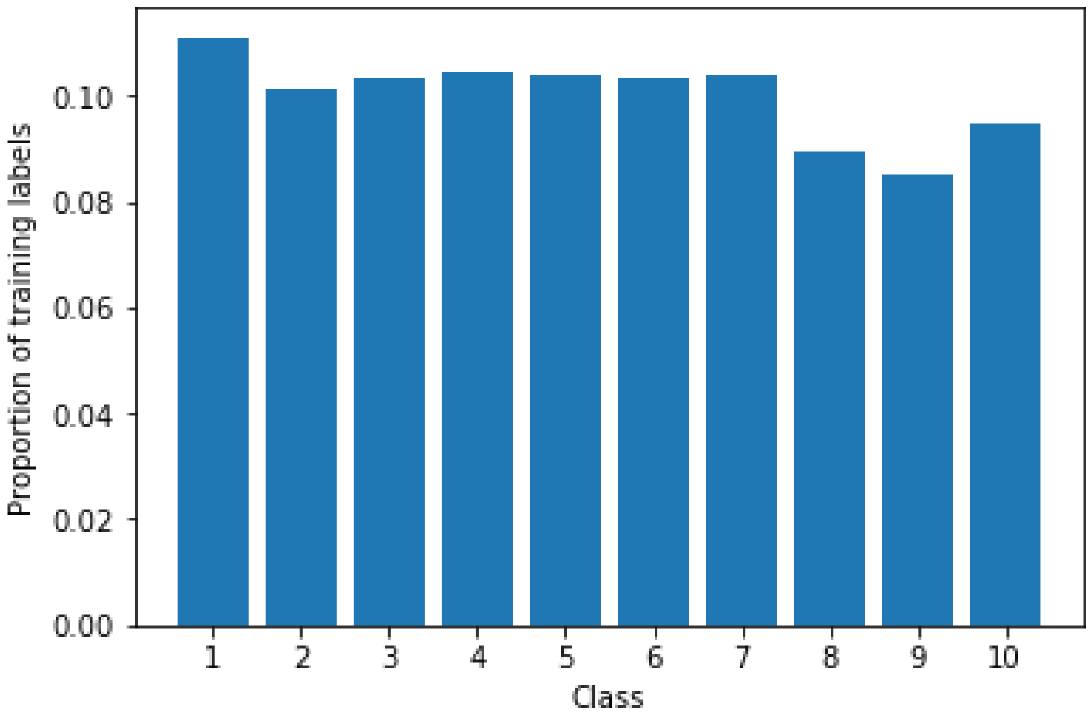
There are also 78,768 unlabelled photos that can be used for testing by submission to Kaggle. Kaggle will provide a multinomial logarithmic loss statistic for our predictions, which we discuss further in the section on performance metrics.

There are 10 classes of activity:

1. Driving safely
2. Texting - right hand
3. Talking on phone - right hand
4. Texting - left hand
5. Talking on phone - left hand
6. Operating radio
7. Drinking
8. Reaching behind
9. Hair and makeup
10. Talking to passenger

Figure 2 shows the frequency of each class label in the training data. We can see this is very balanced, with no class occurring with a frequency less than 8% or greater than 12%. Unfortunately we do not have labels for the larger testing dataset, because it is from Kaggle.

Figure 2: Frequency of class labels



The input images are colour (RGB) and 640*480 resolution, so each image has 640*480*3 = 921,600 features. I decided to re-size the images to 224*224, because I initialised my model with VGG16 weights that had been pre-trained on the Imagenet dataset.

There is 1 unusual characteristic of the training data: while there are 22,424 different images, there are only 26 different drivers. This means that a lot of images are very similar indeed, and using a standard random train-validation split procedure does not work. I discuss this issue further in section 5.1.

3.2 Benchmark Model

I provide two choices of benchmark model.

The first benchmark is a nave classifier that does not require any training. The most common class is safe driving, which constitutes around 11% of all the photos. Therefore, a classifier that always predicts ”safe” will be correct 11% of the time. This will be a good benchmark to judge accuracy against, but it cannot provide a benchmark for loss because it predicts all other classes with zero probability and log loss will be infinite when wrong.

An alternative benchmark is to predict a random class out of 10. This would clearly achieve an accuracy of around 10%, as it will have a 1/10 chance of being right on any given photo. Furthermore this can provide a benchmark for loss as it never predicts with 0 probability.

3.3 Performance Metrics

The first metric I will use is the accuracy. For some classifier $f(x)$, sample with size N and set of labels y:

$$accuracy = 1 - error = 1 - \frac{1}{22,000} \sum_{i=1}^{N} 1 f(x_i) \neq y_i \tag{1}$$

The error is simply the sum of all incorrectly classified images. We would like to minimise this metric by having as few errors as possible. As described above, the first benchmark classifier of always predicting not distracted will have an accuracy of 11%. The alternative benchmark, predicting randomly, will have an accuracy of 10%.

Accuracy is a useful metric because it is easily interpretable. If we report our classifier has an accuracy of 90% then just about anyone can understand that. Furthermore our dataset is relatively balanced, as there is no class which dominates the dataset, which makes accuracy a meaningful metric. The disadvantage of accuracy is that it does not account for prediction confidence i.e. it does not distinguish between a correct but hesitant prediction, and a correct, confident prediction. The latter suggests a better classifier. Another disadvantage is that Kaggle test data is unlabelled, so we cannot calculate accuracy on that dataset.

The second metric I will use is the multiclass logarithmic loss. For sample of size N with 10 classes, labels y and some prediction probabilities given by p:

$$loss = \frac{1}{22,000} \sum_{i=1}^{N} 1 y_{ij} log(p_{ij}) \tag{2}$$

y_{ij} is equal to 1 if the observation i is of class j. To evaluate this metric, our classifier will need to output a probability for each class. We want to minimise this metric, which occurs when our classifier predicts values closer to 1 for the correct class and values closer to 0 for the wrong class.4 We cant calculate the score for our first benchmark model because it outputs zero probabilities for all classes but not distracted, which would lead to an infinitely poor score because of the log function. However the alternative benchmark would have a score of 2.3 (which is equal to log(0.1)).

Log loss has several advantages over accuracy. It performs well in unbalanced datasets, although that is not an issue for this project. It also rewards classifiers based on the confidence of their predictions, as well as the accuracy. Finally Kaggle can compute a loss metric for the test data, which allows us to compare against the validation score. We cannot do this for accuracy.

The main disadvantage of log loss is that it is more difficult to explain to non-technical audiences. It also punishes very confident errors very heavily: as prediction probability tends to zero or 1, log loss tends to infinity if the prediction is incorrect. This may be too harsh on a single mistake.

4 The Neural Network

4.1 General Neural Networks

NNs have become popular for solving complex problems where domain knowledge is limited, such as image recognition. They can pick up complex combinatios of features due to their fully non-parametric structure and massive number of weights.

The smallest unit in the NN is a node. Each node receives some input vector X, applies a weight vector β and produces some output $g(\beta X)$ that depends on the weighted sum of the inputs. The dependence can be non-linear, which is actually necessary for deep neural networks to model very complex relationships, because consecutive linear relationships can always be reduced to a single linear relationship. An example of a non-linear dependency is the Rectified Linear Unit (RELU) activation function, which outputs $max(0, g(X\beta))$. We use the RELU function for all layers in our CNN, because it does not suffer from the "vanishing gradient" problem of other non-linear activations such as Tanh or Sigmoid where the gradient becomes very small as values become large, making it difficult to update the network's weights.

Multiple nodes can be combined to form layers, which are generally square shaped, and multiple layers form the network. The basis structure of a deep NN always consists of:

- The input layer. This is always the first layer and will take raw numerical data e.g. pixels from an image.
- The output layer. This is always the last layer and outputs the predictions of the NN e.g. the probabilities of an image belonging to each possible class.
- The hidden layer(s). Most deep networks have many hidden layers. The first hidden layer's inputs will be from the input layer, and the final hidden layer's outputs will be the inputs for the output layer. Generally earlier hidden layers will focus on picking up simpler features, such as straight lines, and later layers will use the outputs from earlier layers to form more complex features such as body parts.

The NN trains by updating its weights through a "backpropagation" algorithm. The intuition behind backpropagation is relatively simple: if the derivative of the loss function with respect to a given weight is negative then increase the weight because that reduces the loss function. Similarly if the derivative of the loss function with respect to the weight on a node is positive then reduce the weight. The weight update will be bigger for steeper derivatives, and the derivatives are given by applying the chain rule to the loss function through the network. Our initial weights are pre-trained on the ImageNet dataset, rather than random. This makes training much easier because the early layers of weights will already be able to identify basic shapes, such as lines, circles etc.

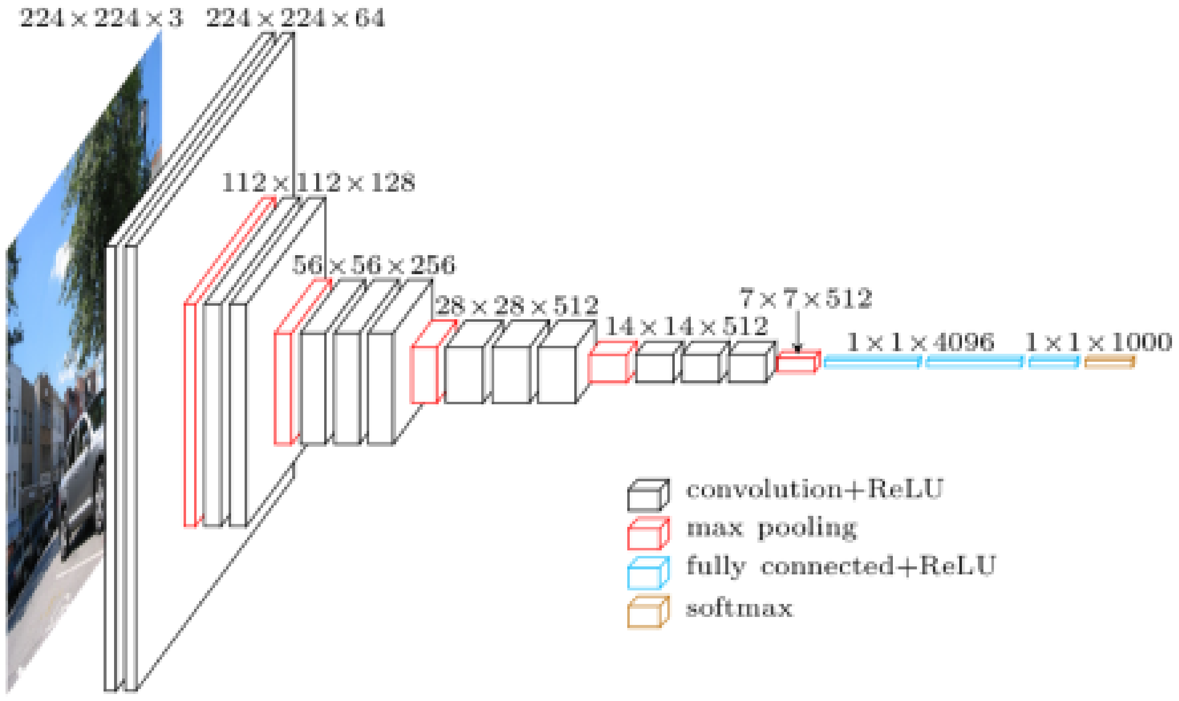
4.2 The VGG16 Network

This project uses a CNN with the VGG16 structure, which is shown in 3. In this section we go through the VGG16 network and explain the purpose of each of the main building blocks.

The first main block is the Convolutional block, which is mostly made up of Convolutional layers. The Convolutional layer is key to any CNN because they allow us to massively reduce the number of weights compared to a standard fully connected layer.

The input to the first Convolutional layer is an image of size 224 (width) * 224 (height) * 3 (RGB). A filter (set of weights) of size 3*3 slides across the input and projects an output for each 3*3 patch of the input. The VGG16 adds zero-padding (an outline of zeros) to ensure the output size is the same as the input size. This

Figure 3: VGG16 Structure



process is repeated 64 times for the layers in the first block, which yields an output size of 224*224*3*64. The key to keeping the number of parameters low is weight-sharing. Each filter shares the same weights when it slides across the layer, so the first layer will have 3*3*3*64 = 1,728 weights. Without weight sharing, the first layer would have 224*224*1,728 = 86,704,128 weights which would clearly become unmanageable as we increase depth. The implict assumption behind Convolution is that only the presence of a given feature in the image is relevant (e.g. a face) for classification, rather than the location of that feature in the image.

There are 4 Convolutional blocks in the VGG16 network and the depth doubles at each block. The early blocks pick up very simple features from the raw data such as straight lines or curves. These simple features are then passed as inputs to later blocks, which can find more complex shapes such as circles, nose outlines etc. The final blocks may pick up complex combinations of shapes which a human could identify, such as faces. Later blocks are deeper because there are more complex features than simple ones.

In between each block of Convolutional layers is a Max Pooling layer. This slides a 2*2 filter across each depth slice of the Convolutional layer and outputs the maximum value in the 2*2 patch, so there are no weights in this layer. The purpose of the Max Pooling layer is to reduce the height and width of future Convolutional layers, which it achieves by taking a stride of size 2 i.e. no patch will share any of the same features. Therefore while the depth of each block doubles, the height and width are roughly halved.

The final block is comprised of Fully Connected (FC) Layers, which are the same as in a regular NN. The FC layers determine which of the complex features outputted by the final convolutional block correlate most to each of the classes. The majority of the weights in VGG16 are in the first two FC layers. These have size 4096, so the first layer alone has 103M weights (input size of 7*7*512 x 4096 neurons), out of the 138M weights in the network. This is vastly more than the final convolutional layer, which has only 2.4M weights. We therefore test a versioin of the VGG16 network without the FC layers. To prevent overfitting, we use Dropout layers from Srivastava et al. [2014] between the FC layers. The Dropout layers randomly set 50% of outputs to zero during each training batch, forcing the CNN to learn to provide the right classification without always relying on certain features.

The output layer is a FC layer with a Softmax activation function. Softmax outputs 10 probabilities, 1 for each class, such that $\sum_{class=1}^{10} P(class) = 1$.

5 Results

5.1 Pre-processing

The first step in pre-processing our data is to convert from 640*480 RGB images into 224*224*3, which can be used with VGG16 weights that are pre-trained on the Imagenet dataset. This was relatively simple: Python's OpenCV package converts images to a given size very quickly. The result is an array of shape (N*224*224*3), where N is the number of images. Each element of the array takes a value between 0 and 255, representing their position on the RGB spectrum.

The second step in preprocessing was to divide all features by 255. This normalises features so that they take a value between 0 and 1, because CNNs train best on features in this range. This step was complicated by memory limits. Consider the testing data: there are around 80,000 images with 224*224*3 features. Given that each integer features takes 4 Bytes of space, the testing data is around 12GB, which was too large to fit on my 8GM ram. I therefore used Numpy's Memap class to process the data in batches. Memap allows for temporary access to sections of a larger file, so the only data stored in RAM would be the current batch that is being scaled down by 255.

5.2 Training Procedure

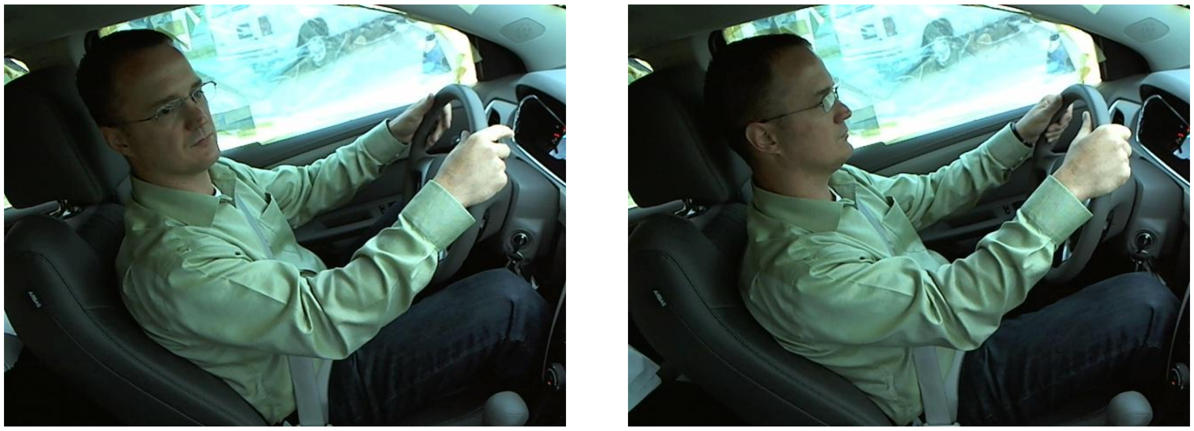
I initially followed the standard procedure of splitting the training data into training and validation sets. After each epoch the CNN is tested on the validation set, which is kept separate from the training set. The CNN stops training when the validation loss stops declining, because that signals the CNN is beginning to overfit. The test data is completely separate, unlabelled and tested by submission to Kaggle, so we do not need to hold out a separate test set from our training data.

I used the Keras wrapper over Tensorflow to train the model. There is no performance loss from Keras but it is much simpler to use than Tensorflow because its syntax follows that of Ski-kit Learn, which is a popular library for Machine Learning. The memory issues (described in section 5.1) applied to training too, therefore I used Keras' fit_generator method to fit the model. I trained using a batch size of 32 and fit_generator ensures that only the current batch is stored in RAM, rather than the whole training dataset.

It was impossible to get an accurate validation loss using a random train-validation split. This is because the dataset's structure is unusual. There are 22,424 training images but only 26 different drivers, so there are many images that are very similar. The CNN may fit to the features of the drivers, rather than the actions they are taking. If the validation set contains images of the same drivers as the training set, then the CNN will get a very good validation score because it has seen these drivers before, despite the images being slightly different.

Figure 4 shows an example of this problem. The two images of the same driver performing the same action are very similar, so if one of the images was in the training set and one in the validation set then we would expect the CNN to classify accurately because it has essentially seen this image before. Therefore the validation loss would not provide a good measure of the prediction capability against unseen drivers. Without an accurate validation loss, I could not know when to stop training the network. Therefore early submissions to Kaggle had high loss scores (nearly 2), because the CNN trained for too long and overfit to the specific features of the drivers, rather than learning to recognise generalisable actions.

Figure 4: Example of very similar images

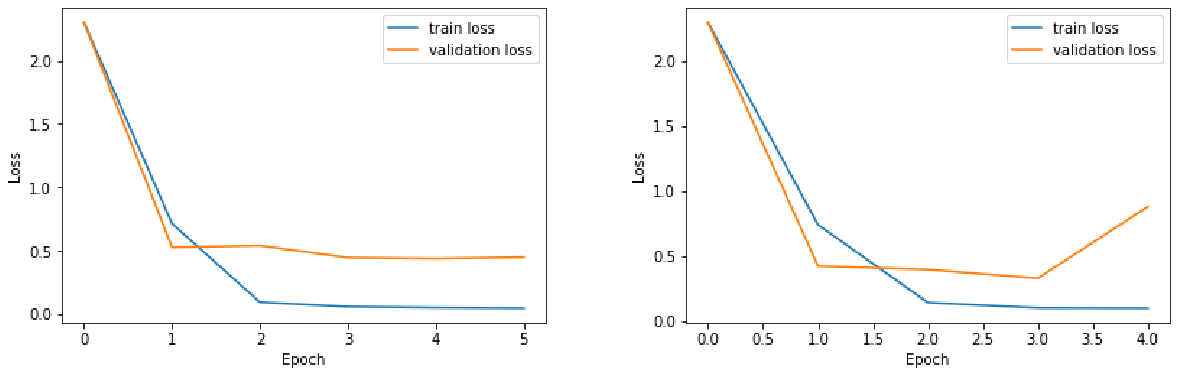

(a) Image of driver going safely (b) Very similar image of driver going safely

5.3 Improvements

To deal with this problem, we construct our validation set non-randomly. Kaggle provides a dictionary that maps each driver in the training data to a list of their images. I chose 5 drivers, roughly 20% of the labelled data, and held out all images from those drivers as the validation data. This meant the CNN would never see 2 images from the same driver in both the training data and the validation data, and the validation score would provide a good measure of when to stop training our model.

Initially the CNN still overfit quite fast, after 1-2 epochs. The default learning rate in Keras was 0.01, with no decay. I therefore introduced a decay of 0.00001 per batch. This reduced the speed of overfitting and we were able to train the final models for 3-4 epochs before validation loss started to rise. I stopped training at the first epoch where validation loss rose - figure 5 shows the losses at each epoch for each of my two final models.

Figure 5: Losses for each model



(a) Base Model (b) No FC Model

Table 1 presents final results from training the CNN. The first column is our base VGG16 model. The second column is the model without the two 4096*4096 FC layers at the end.

Table 1: Results

Metrics	**Model**	
	Base	**No FC**
Epochs	4	3
Train Time	270m	129m
Train Acc	0.994	0.985
Train Loss	0.0514	0.098
Valid Acc	0.862	0.881
Valid Loss	0.439	0.328
Test Loss	0.443	0.343
Predict Time	0.1s - 0.12s	0.1s - 0.14s

Table 2: Learning rate = 0.001, decay = 0.00001

All models improve greatly on the benchmark of 10% accuracy. The base model has a validation accuracy of 86.2% and the model without the FC layers has validation accuracy of 88.1%. However we cannot evaluate accuracy on the test data for Kaggle submission, because it is unlabelled and Kaggle only provides a loss metric.

All models also improve markedly on the benchmark loss of 2.3. We see that testing loss is only slightly higher than validation loss in both models, so our method of holding out drivers was successful. However testing loss greatly exceed training loss suggesting that both models overfit quite quickly after just 3-4 epochs. The training losses are unrealistically low, implying accuracy of 99.4% for the base model and 98.5% for the model without the FC layers.

The model without the FC layers performs slightly better on the validation set and significantly better on the test set, with a loss of 0.343 compared to the base model's test loss of 0.443. This is probably because the FC layers contain most of the weights so removing them can help reduce overfitting. Removing the FC layers also helps with training times, which are about half as long for the model without the FC layers.

The models take a long time to train because I only have access to a NVIDIA 880M GPU. Each epoch is at least 43 minutes for the model without the FC layers, and epochs are around 63 minutes for the base model. However the test times are fairly good - both models can predict any given image in a fraction of a second. This would be fast enough for use inside cars as part of a warning system to distracted drivers.

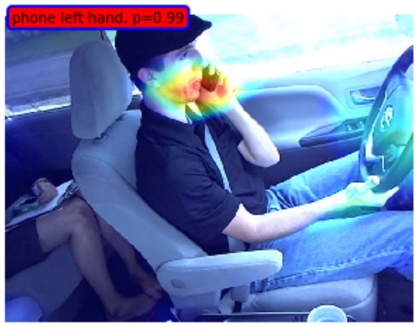
5.4 Interpretation

A common criticism of neural networks is that their decision process is a black box. We often have little idea why a neural network makes a certain prediction, even though it is very accurate. We tackle this problem using a Gradient-based Class Activation Map (GradCAM - see Selvaraju et al. [2016]). The GradCAM produces a "heatmap" that provides insight into how the CNN is making decisions. "Hotter" areas of the image get more weight in the decision. The GradCAM has the further advantage of being able to deal with models with FC layers, so there is not necessarily a trade-off between performance and interpretability.

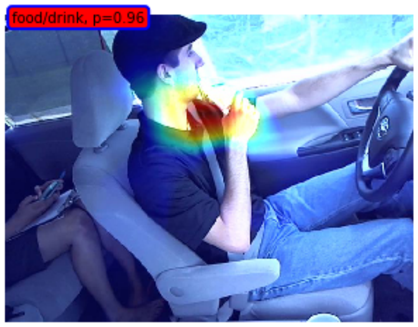
Figure 6 shows some heatmaps for a variety of classifications. We can immediately see that the CNN focusses on quite a small section of the images. For example, figure 6a shows that it learns to classify whether a driver is talking on the phone by looking only at the region around the ear and nothing else. figure 6b shows a similar pattern for eating/drinking, except the CNN looks only at the area around the mouth. We can also see from figure 6d that the CNN learns to recognise some objects - it very confidently correctly predicts that a driver is texting by picking up on a phone in the image.

The heatmaps give us some insight into where the CNN predicts less accurately. Figure 6e shows an example of a misclassification. The CNN predicts that the driver is eating/drinking, when actually they are probably checking their appearance. It's easy to see why the CNN makes this mistake - the driver has their hand raised to their face in the same manner that they would if they were eating / drinking, however the CNN has not learned to detect whether there is an object in the hand. A mitigating factor is that the CNN is not that confident in its prediction, with only 71% probability.

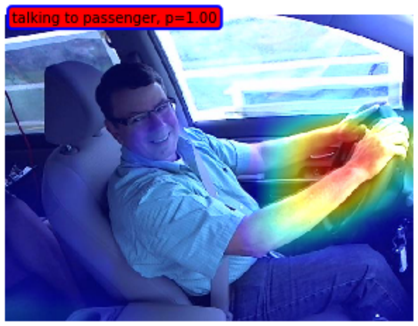
Figure 6: Examples of GradCAM



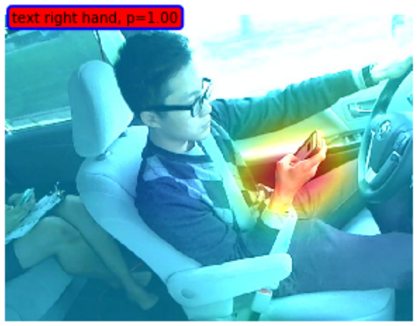
(a) Correct prediction



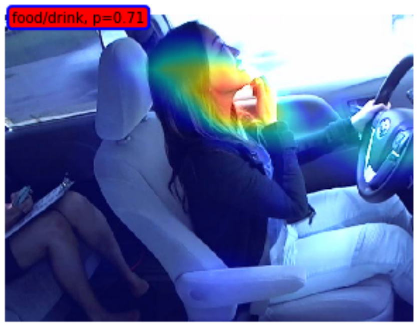
(b) Correct prediction



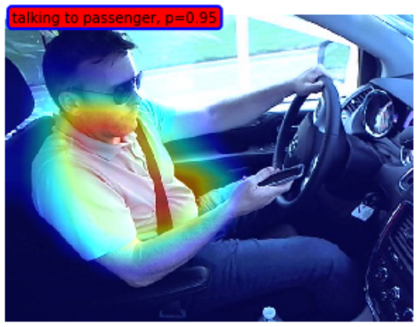
(c) Possible overfitting



(d) Correct prediction

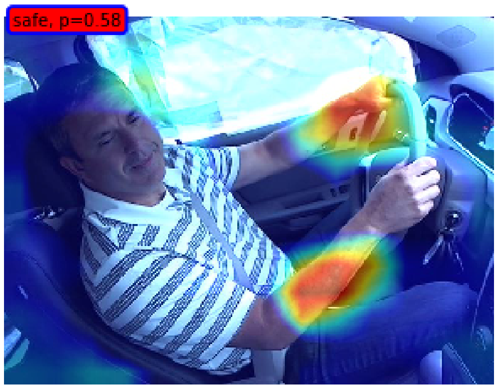


(e) Wrong prediction

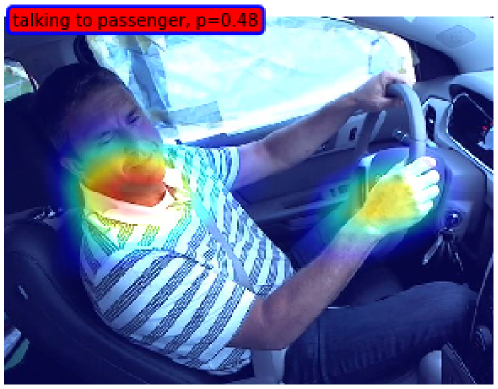


(f) Wrong prediction

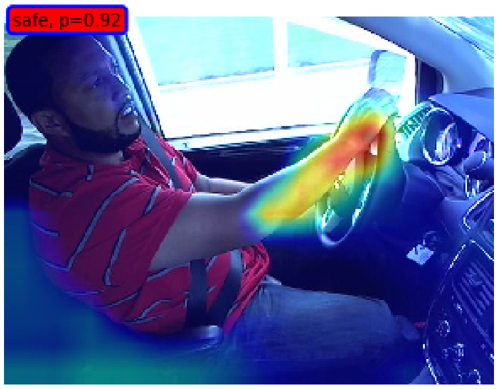
Figure 7: Model comparison using GradCAM



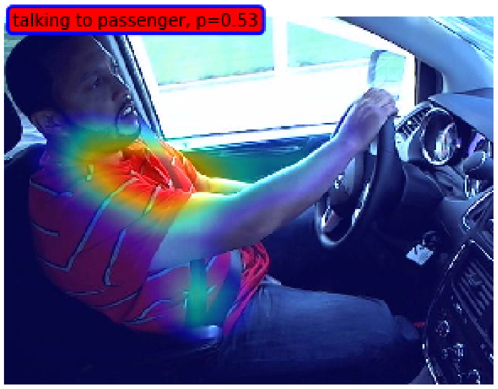
(a) Base model



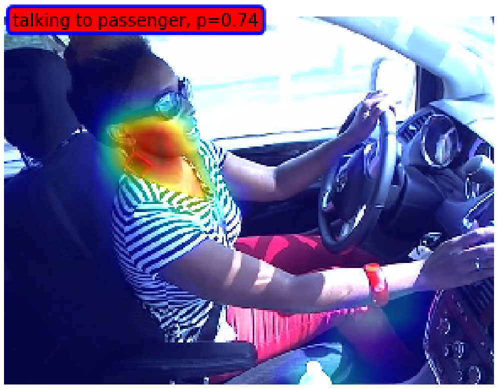
(b) No FC model



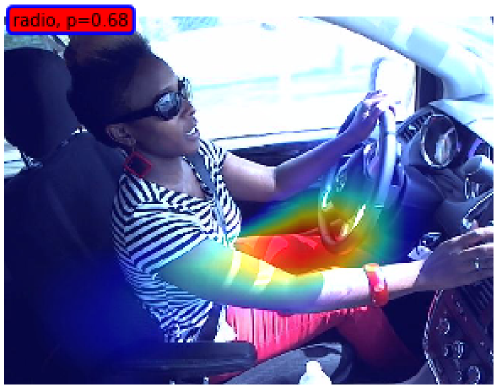
(c) Base model



(d) No FC model



(e) Base model



(f) No FC model

We also get some insight into why the CNN overfits from the heatmaps. Figure 6c is a training image, where the correct classification is talking to passengers. The CNN predicts this with 100% probability, but does so by focussing on the drivers' hands which are on the wheel. This seems like overfitting as this would be consistent with safe driving too. Figure 6f shows an incorrect prediction in the test set. The CNN quite confidently predicts (95%) the driver is talking to a passenger when they are in fact texting. The CNN is probably focussing on the wrong area.

A final use of the heatmaps is to compare the models. We think the heatmaps provide additional evidence to the performance statistics in showing that the model without the FC layers tends to overfit less. We show this using the images in figure 7, where the first column is the model with FC layers and the second column is without FC layers.

Generally the models without FC layers predict less confidently, which is helpful when the model makes mistakes. This is evidence of avoiding overfitting. For example figure 7c and figure 7d show a difficult prediction where the correct outcome could plausibly be either driving safely or talking to a passenger. The base model predicts driving safely with 92% probability, whereas the model without FC layers predicts much less confidently with 52% probability.

There's also some evidence that the model without FC layers has learned to focus better on the correct areas. In figure 7e, the model with FC layers makes a wrong prediction by focusing on the mouth, rather than the arm which is clearly adjusting the radio. Figure 7f shows that the model without FC layers correctly picks this up.

6 Summary

6.1 Workflow

In this project I aimed to build a classifier that could accurately classify drivers' images by whether or not they were driving safely. I sourced the imaged data from Kaggle and transformed the images from 640*480 pictures to a 224*224*3 numerical array. There were many programming challenges, in particular memory was often a binding constraint. I had to extensively use batches to both process data and train the model. Numpy's Memap class was very useful for storage - it allowed me to temporarily store in RAM only the batch that I was currently processing or training on.

Results were initially poor because of the unusual structure of the dataset, which had many similar images from the same drivers. The usual random train/validation split did not provide a good performance measure on the validation set, because many of the images were very similar to those in the training set. Validation loss was inaccurate and I did not know when to stop training, which lead to overfitting and very poor test scores. I dealt with this issue by holding out all images from one fifth of the drivers in the training data as the validation set, therefore providing an accurate loss measure so that I knew when to stop training the network.

The final results were satisfactory. By building a CNN with a modified VGG16 structure, I was able to make correct predictions over 10 classes with accuracy of 88%. This is clearly a large improvement over the benchmark classifier, which would have accuracy of just 10%. These results generalised well to the unlabelled Kaggle testing data, where I scored a log loss of 0.34. This was actually slightly better thanp the validation loss. Furthermore prediction times are quick, at roughly 0.1 seconds, which makes the model suitable for use inside a car's real-time warning system.

6.2 Reflection and Suggestions

I made some progress towards opening the "black box" of the CNN's decision process, by using the GradCAM method to create "heatmaps". These showed us where the CNN was paying most attention when making its decision, and allowed us to see why it did better on some types of images than others.

I find that removing the fully connected layers from the base VGG16 structure can help to deal with overfitting. I reported an improved test loss, and further support this conclusion with heatmaps showing why the fully connected layers can lead to overconfident and incorrect decisions.

I was limited by computational power so was unable to fine-tune the CNN was much as I would have liked. However I can still suggest some improvements. Firstly, future projects could experiment with adding more convolutional layers such as the VGG19 structure, or a different architecture altogether. Second, more data from different drivers would improve the CNN. Whilst I had over 22,000 training images, there were only 26 different drivers and I was unable to prevent the CNN from overfitting to the features of particular drivers, rather than looking for features that generalise well to new drivers.

References

C. for Disease Control and Prevention. Distracted Driving kernel description, 2017. URL `https://www.cdc.gov/motorvehiclesafety/distracteddriving/index.html`.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, Dec. 1989. ISSN 0899-7667. doi: 10.1162/neco.1989.1.4.541. URL `http://dx.doi.org/10.1162/neco.1989.1.4.541`.

M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

U. S. D. of Transportation. Distracted Driving kernel description, 2017. URL `https://www.nhtsa.gov/risky-driving/distracted-driving`.

R. Raina, A. Madhavan, and A. Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 873–880, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-516-1. doi: 10.1145/1553374.1553486. URL `http://doi.acm.org/10.1145/1553374.1553486`.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X. URL `http://dl.acm.org/citation.cfm?id=104279.104293`.

R. R. Selvaraju, A. Das, R. Vedantam, M. Cogswell, D. Parikh, and D. Batra. Grad-cam: Why did you say that? visual explanations from deep networks via gradient-based localization. *CoRR*, abs/1610.02391, 2016. URL `http://arxiv.org/abs/1610.02391`.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL `http://arxiv.org/abs/1409.1556`.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, Jan. 2014. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=2627435.2670313`.

P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

B. Zhou, A. Khosla, L. A., A. Oliva, and A. Torralba. Learning Deep Features for Discriminative Localization. *CVPR*, 2016.